Business Process and Software Architecture Model Co-evolution Patterns

Pooyan Jamshidi, Claus Pahl
Lero - The Irish Software Engineering Research Centre
School of Computing, Dublin City University,
Dublin, Ireland
[pooyan.jamshidi, claus.pahl]@computing.dcu.ie

***Abstract*— Software systems are subject to change. To embrace change, the systems should be equipped with automated mechanisms. Business process and software architecture models are two artifacts that are subject to change in an interrelated manner that requires them co-evolve. As opposed to the traditional batch-based model transformation, we propose a comprehensive set of structural and behavioral evolution patterns that enable to incrementally reflect the impact of change of business processes to their associated architecture models by applying reusable patterns. A basis for automation is provided through a graph-based formalism.**

***Keywords*- software architecture; co-evolving models; evolution pattern; graph-based model evolution**

I. INTRODUCTION

Software-intensive systems are subject to changes, usually driven by external stimuli from the environment [1] as diverse as technological changes or reengineered business processes. To cope with these issues, software artifacts produced and used by the software-intensive systems have to evolve. Depending on the artifact type and granularity, the impact and rate of change may differ. We concentrate on interrelated changes that happen between business process models and their supporting software architecture descriptions. Business processes convey the requirements of today's process-aware software systems and they are tightly coupled with underlying software architectures. This type of change requires creating multiple modeling elements and connections among them in both business processes and software architecture description. Manual change not only affects modeling performance, but also model correctness [8]. To support change management, it is crucial to identify different types of changes (at the process layer) and change impact patterns (at the architecture layer). Both of these evolution categories and their impact on each other would benefit from extracting change patterns reoccurring during evolution and provide the promising basis for automation.

How heterogeneous modeling languages semantically fit together or how to consistently co-evolve still has challenges [1]. This is especially true for business process and software architecture models which are interrelated implicitly by set of architectural decisions. To put the problem into an abstract perspective, consider the conceptual model of the software architecture co-evolution problem in Fig. 1. There exists a mapping between the process model (P) and software architecture model (A). This mapping (F) embraces the initial architectural decisions that are represented as a set of structure-preserving functions between the two associated models. This part of the conceptual model is being addressed in model-driven research and is utilized here. We provide mechanisms that preserve these structure-preserving mappings during the evolution of software architecture. Business process and workflow management communities have been investigated the process model changes by identifying a comprehensive catalogue of change patterns [9]. We assume that the transformation function (T) is given as a set of business process change patterns. Therefore, the main focus here is on how the software architecture model can be adapted to the changes raised by process models with an emphasis on preserving initial architectural decisions.

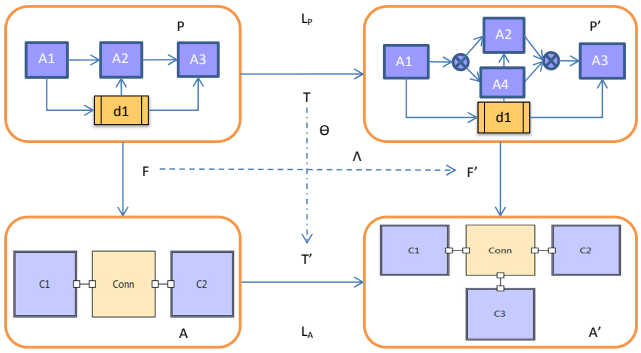

Figure 1. Software architecture co-evolution conceptual model
P, P', A, A': Model; T, T', F, F': Model Evolution (Transformation); and
Λ: Transformation Evolution; Θ: Change Impact

After some background in Section II, we define two levels of interrelated models, based on which a set of change impact and mapping change patterns to enable architecture evolution (Section III) and a formalism supporting its automation (Section IV) are defined. The key implication of these patterns is that they facilitate the change analysis and can be reused in the evolution process, illustrated in Section V, before ending with related work and some conclusions.

II. BACKGROUND

A change to a model in a model-driven system may require other changes to be made to other interrelated models, referred to as *consequential change*. If the reason for a consequential change can be formalized, then its derivation and application can be automated [13]. A reason for consequential change in a model-driven system is the *preservation* of mapping relationships, which are representing architectural properties. The other consequential changes require specialized domain knowledge and may not be automated without that embedded knowledge [13, 14].

Traditionally, model transformation tools support the batch execution of transformation rules, which means that input is always processed "as a whole", and output is always

regenerated completely [13]. However, for interrelated models which are subject to change continuously, batch transformations with some flavors of model matching mechanisms (ex. AML) may not be suitable and need to be replaced by incremental model transformations [13] (change-driven transformation [21]) to update existing target models based on changes in source models and to minimize the parts that need to be reexamined by a transformation when the source model is changed. Moreover, in some occasions only an external interface is available for query and manipulation of models making them non-materialized [21]. In addition, *traceability information* can also be limited and externalized, which imposes further challenges [13]. Finally, certain constraints can be evolutionary in the sense that they need to be evaluated over a sequence of model evolution steps and not over a single snapshot of the model [21].

There are structural elements in software architecture that have been defined [12]. We also consider behavioral aspects (Fig. 2) that describe the protocol of the entire architecture configuration, or component or connectors specifying interactions between the elements. Component behavior defines interactions of the component with its environment as sequence of operations through its interfaces (protocols).

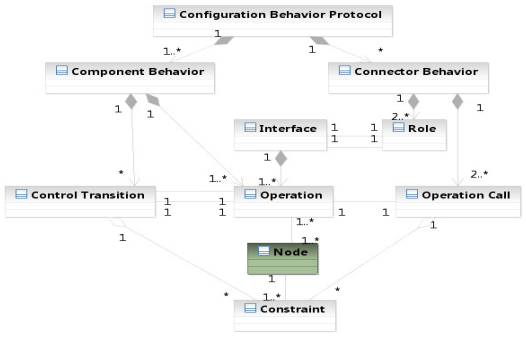

Figure 2. Software architecture behavior specification meta-model

A change impact pattern is a reusable source of knowledge concerning the co-evolution of two related artifacts. The primary changes in a given artifact (called driver) are characterized via a change scenario. In order to cope with this change, a change pattern provides mechanism how to adjust the second artifact (called companion) [7]. We focus on process models as driver and architecture models as companion. When the process model of a system evolves, most likely the architecture model needs to be updated too.

III. Software Architecture Evolution Pattern

Instead of discussing primitive changes, such as adding or removing a primary element, discovering and utilizing different types of semantics-aware changes (changes that reflect the intention of change) allow for better manipulation and consequent adoption [9, 15]. We present, firstly, the various changes of business process and their impact on software architecture called change impact patterns (denoted by Θ-transformation in Fig. 1) and, secondly, the change patterns that have been captured in order to reflect different changes to the mapping between business processes to architecture models (Λ-transformation in Fig. 1).

We consider patterns identified and specified in business process and software architecture evolution. We focus on the evolution of software architecture as a reaction to changes of the business processes. We first created a list of candidate patterns in either of the domains based on a literature review and experimental work. For business process changes, we adapted the change patterns proposed by Weber [9] slightly to meet our needs. We compared the available as-is and to-be processes from different business process instances and their associated software architecture models.

A. Change Impact Patterns

We have defined seven change impact patterns of software architecture as a consequence of change scenarios in business process. Each change impact pattern captures a specific type of change effect. The transformation rules are informally presented in using an activity diagram to show business process and component behavior, sequence diagram to show connector behavior and later in Section B, xADL description language to model architecture, which is more common in practice and easier to understand. Instead of using general notations, a concrete syntax allows describing the same behavior using concepts closer to the area of business process and software architectures (e.g. activity, flow, components, and ports).

Change pattern 1 - embed activity in conditional branch: Depending on the location of a conditional branch, two software architecture changes are possible: a constraint needs to be added to the behavior protocol of a connector (i) or to the behavior of a component (ii). Fig. 3 ((a), change in connector behavior protocol which is specified by a sequence diagram) (resp. (b), change in component behavior protocol which is specified by an activity diagram) shows an example of the change impact (i) (resp. (ii)). After the process change, an activity in a conditional branch that is located between two activities related to a component different from (same to) the component whose behavior is associated with this embedded activity. This process change has an impact on the connector behavior (resp. component behavior) protocol, see Fig. 3 (a) (resp. (b)) with constraints derived from the conditions of the corresponding *xor* branch.

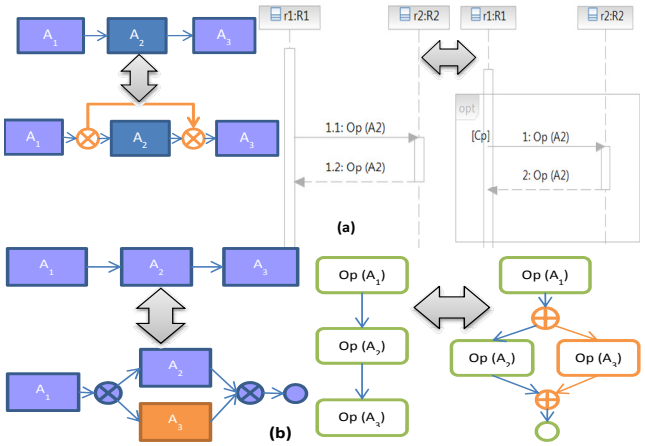

Figure 3. Embed activity in conditional branch

Change pattern 2 - move an activity (serially, parallel, conditionally): Depending on the business process changes including moving activities, parallelizing activities, and sequencing activities, three types of change impact are possible as follows: (i) transition sequences in component behavior must be reordered; (ii) sequential transition sequences must be changed to parallel transition sequences in component behavior; and (iii) parallel transition sequences must be changed to sequential transition sequences in component behavior. Fig. 4 shows an example with two activities parallelized and its impact on an associated component behavior specification.

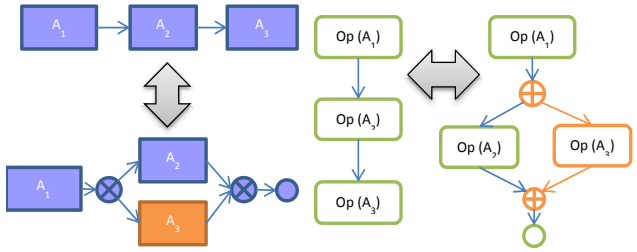

Figure 4. Parallelize an activity

Change pattern 3: insert an activity (serially, parallel, conditionally): Depending on the process changes that insert an activity, two types of change impact are possible as follows: (i) a new component and connector must be added; (ii) an operation needs to be added in component behavior.

Due to lack of space, we only name the other change scenarios: (4) *insert an activity between two individual activities*: if these two activities embedded in a condition then this condition and associated operation with the activity should be reflected to a component behavior, (5) *replace an activity*: the operation in the component behavior associated with this evolved activity should be updated, (6) *update a condition*: the condition in the component behavior associated with this evolved condition should be updated, and (7) *embed an activity in a loop*: the operation associated with this activity in component behavior should be embedded with a loop and its condition derived from the loop reflected in business process.

B. *Mapping Change Patterns*

The mapping between business process and architecture description comprises decisions that have been made by architect. These architectural decisions act as traceability links between business process and architectural elements. In this section, we list types of change patterns that might happen to these mappings. These changes have been mainly extracted from [16] and adapted to hierarchical architectures.

The initial configuration of the example consisting of two components (C1 and C2) which are connected by a connector (Conn) as depicted in Fig. 5.

Figure 5. Initial software architecture configuration

Abstraction factors out a part of the configuration by encapsulating it within an instance of a proper architectural element (in this case the component C in Fig. 6).

Figure 6. Architecture model after applying "abstraction"

Extension involves adding new interface ports, components, and connectors to a given configuration. In order to apply an extension to a configuration, the sets of new architectural elements (C3 and Conn in Fig. 7) must be specified.

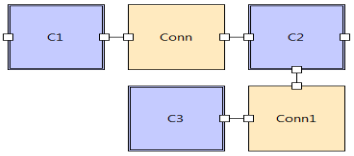

Figure 7. Architecture model after applying "extension"

Refinement involves extending the existing architectural elements in the configuration by preserving external interfaces of the changed elements (C2 in Fig. 8 by adding two subcomponents and their corresponding binding).

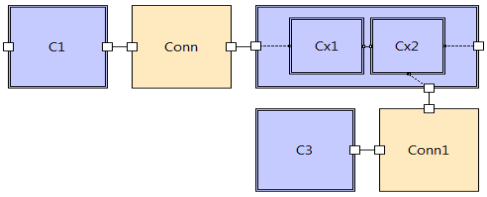

Figure 8. Architecture model after applying "refinement"

Flatten unravels a configuration with respect to a constituent element (C2 in Fig. 8). In the resulting architecture model, the element is replaced by its own components and connectors (Cx1 and Cx2 in Fig. 9).

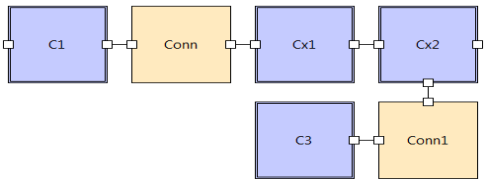

Figure 9. Architecture model after applying "flatten (wrap)"

Rewire reconfigure an existing configuration by changing the connection between component and connectors and constructs the new configuration with the same elements but different combinations, shown in Fig. 10.

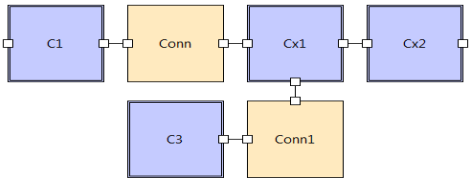

Figure 10. Architecture model after applying "rewire"

Replacement substitutes an arbitrary architectural element with a new element (Cx2 in Fig. 10 with Cy2 in Fig. 11).

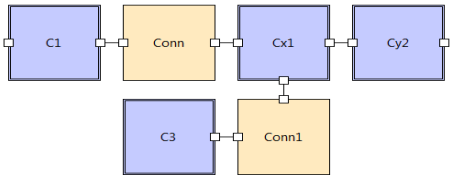

Figure 11. Architecture model after applying "replacement"

Formally, these structural operators form a partial function on structures. When an operator is applied to a structure that satisfies the preconditions, it returns a new structure together with a semantic correspondence function from the interface ports of the operand to those of the resulting structure. More specifically, these change patterns are homomorphisms that map the structure of an architecture model to that of a second variation, while preserving both the internal and the external configuration of the first. The notion of a homomorphism can be used to express the fact that a given architecture model may structurally evolve into another model such that for each interface, component, and connector of the first model, the second model has a corresponding interface port, component, and connection, respectively, with a similar (or larger) semantic role [16].

IV. Formalization of the Co-Evolution Process

As described in Section III, an evolution process comprises the execution of both change impact patterns and mapping change patterns where each evolution pattern has an impact on the description of an architecture. The change patterns can be formalized as typed graph transformations [10], supported by a sound theory and a variety of tools for its execution and analysis. The key reasons to select typed graph transformations are as follows: (i) business process and software architecture models can be easily formalized as graphs, as shown in other works [11]; (ii) graph transformation rules are self-contained and independent of each other that benefits the formalization of pattern-based evolution because each rule can be applied when its preconditions are satisfied and reused several times to make the required changes in their respective models as traditional patterns do; and (iii) typed graphs capture the relation among business process and architecture types, required to transfer the changes between the models.

A graph-based formalism to support the application-level notation provides a sound basis on which it is possible to develop automated applications. With graph-based transformation rules, the change history of business process and corresponding software architecture can be recorded as in Fig. 12. A business process model (BPM_i) is mapped to its successor ($BPM_{i'}$) and architectural model (ADM_i) is mapped to its successor ($ADM_{i'}$) by a set of transformation rules (T^b_{ij} and $T^a_{ij,}$). The mapping between business and process models is shown by dashed arrows. Thus, the transformations represent changes that establish semantic correspondences between neighboring pairs of models along a given path in the evolution history graph and the mappings represent structure-preserving correspondences between neighboring pairs of models [17], see Fig. 12. The evolution history graph can be rolled back in order to undo the effect of the applied transformation rules. For instance, if we want to roll back to the ADM_3 we need to automatically undo the effects of "T^a_{i4}", "T^b_{i2}, and T^ai3" to reach the "BPM_7-ADM_4" and "BPM_1-ADM_3" respectively from ADM_7.

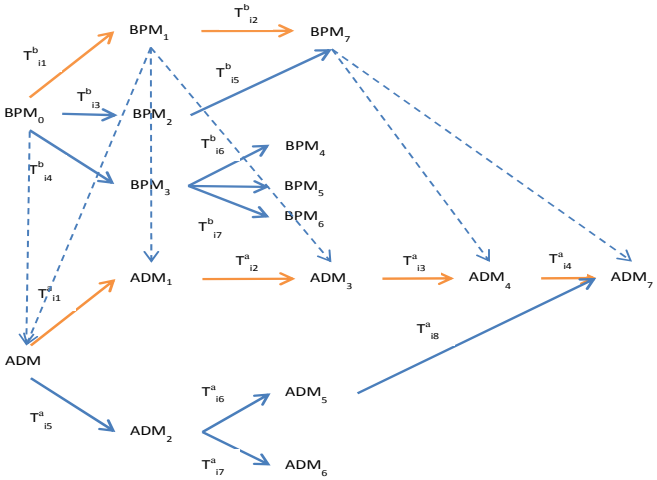

Figure 12. A co-evolution history graph

A graph-based foundation facilitates analytic functions. We reuse the edit-distance metric proposed in [9] to measure the number of change operations minimally required to transform an architecture model ADM_0 to an evolved counterpart ADM_1. By calculating the edit distance, the complexity of transforming one model to another one can be measured and by summing up these measures along a path in the evolution history, the complexity of a composite transformation can be determined. In general, when changing models using change patterns instead of change primitives, the edit distance can be decreased. The architecture model in Fig. 5 illustrates a configuration consisting of two components (C1 and C2) and a connector (Conn). Assume that a structural change should be accomplished inserting an additional Component C3 (Fig. 7). The change pattern "Extend" provides the high-level change operation Extend (ADM_0, C3, Conn1), which allows users to add component C3 and connector Conn1 into the current configuration. In the example, the transformation of the original architecture model into the evolved counterpart requires eight change primitives (adding *1 new component*, *1 new connector*, *4 new ports*, and *2 bindings* to connect them), resulting in an edit distance of eight. Using the extend pattern in this example, seven (primitive) operations can be replaced by one (high-level) change pattern. Depending on the structure of an architecture model, the implementation of a change pattern with change primitives can result in different edit distances. The edit distance also depends on the meta-models and the adopted tool. Although the edit distance does not allow for quantifying how much time is needed to accomplish a respective change, it allows evaluating the number of copy effort/steps needed.

V. APPLICABILITY OF THE EVOLUTION PATTERNS

This section demonstrates our approach using a case study. We aim to illustrate how adopting change patterns introduced in this work are *useful* for analyzing and applying the impact of change of business processes to corresponding software architecture models and can support *automation* in this adaptation process.

In this case study adopted from [17], a scenario of integrated loan management (LM) services is investigated. The main process is a loan management process. The aim in this case is to investigate the interrelationships between the business process and software architecture model and also the impact analysis of some changes in the business process model by utilizing the introduced change patterns.

Fig. 13 shows a simplified *loan management process model*. The process starts when a client requests a loan by email. A bank agent calls the client to present an offer. To provide the offer, the agent needs to obtain client data, calculate the amount of loan offer and call the client to explain conditions of the loan. If the client accepts the offer, then a direct sales agent located near the client would visit her. After the contract is established, she registers the visit with her signature and reports the visit to her supervisor.

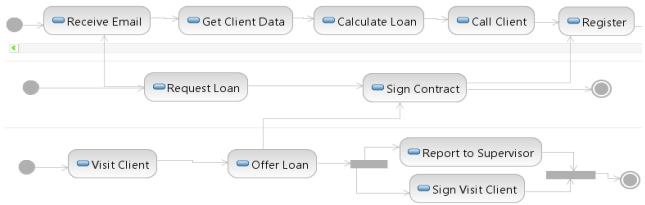

Figure 13. BPM_0: The initial LM process model

The *software architecture model* supporting the process is shown in Fig. 16. The involved components are a mail server ("Email-mng"), a customer relation management component ("CRM"), an application managing the client's bank account ("Account"), and two applications managing the information of sales and plans of the bank, "Sales" and "Planning", resp. The functions exposed by each component are isolated and the interaction between functionalities during the process is managed by the sales agent.

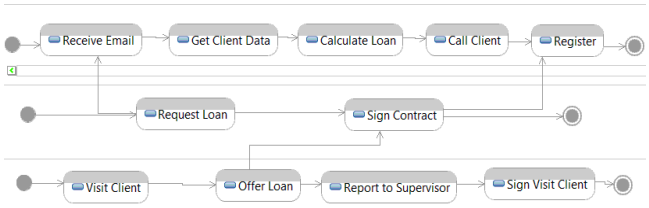

Figure 14. BPM_1: evolved LM process model after change (move activity)

The *change scenarios* involve modifications resulting from the improvement of a bottleneck activity that affected the operation of the business, the incorporation of new process regulations and technological updates. For the first business change scenario (BS1), stakeholders at the bank asked architects whether they could facilitate the tasks of the sales agent in order to have mobile access to banking application functionalities. They suggested integrating the software supporting the agents work to allow remote reporting to the supervisor using a mobile device. Thus, she is only required to sign a physical document at the office. As a consequence of this business change, the initial process model is modified to the process illustrated in Fig. 14.

For the second business change scenario (BS2), the "get client data" activity becomes optional for those instances that have customer data enclosed. This change is reflected in the business process by embedding this activity in a conditional branch as shown in Fig. 15.

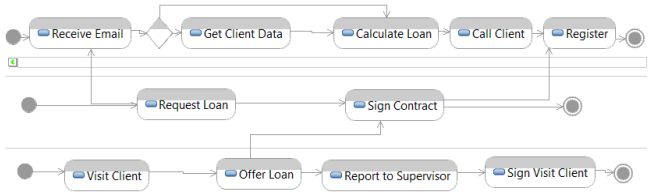

Figure 15. BPM_2: evolved LM process model after change (embed an activity in conditional branch)

The *technical change scenarios* show the evolution requirements and the designated solutions to adopt accordingly. The technical change scenarios are as follows:

1. (TS1) Integrate the applications supporting the "Calculate Loan Offer" activity
2. (TS2) Integrate all the applications in a reliable way that keeps the applications independent from each other
3. (TS3) Adopt service architectural style for integrating the services exposed form applications

In order to respond to BS1, i.e. *apply of the change patterns on the loan management change scenarios,* the change moves two parallel activities into sequence. These activities are a part of the behavior exposed through the Sales component in the architecture model, change pattern 2 (move an activity) is utilized. In the evolved architecture model, the behavior protocol specification of the Sales component is affected accordingly.

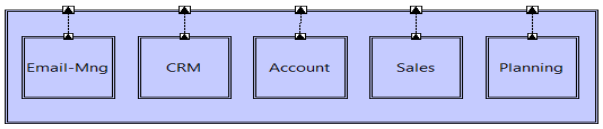

Figure 16. ADM_0: The initial LM architecture model

For BS2, since the change embeds the "get client data" activity in a conditional branch and the functionalities of this activity and the next "calculate loan" activity has been implemented by CRM and Sales integration with Planning components respectively, the change should be reflected (by utilizing change pattern 1) in the behavior protocol of the connector facilitating this integration.

As the consequence of TS1, the Planning and Sales components are to be connected (Fig. 17). Therefore,

rewiring pattern should be adopted in order to make an appropriate change in software architecture description.

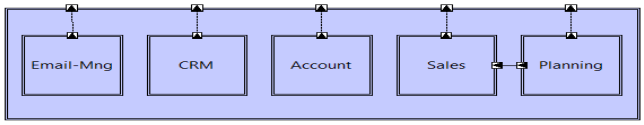

Figure 17. ADM_1: evolved architecture by applying rewire

In order to integrate the remaining applications, a messaging schema and a central component (CMM) to persist messages was added. A composed functionality was designed to serve the Loan to Client activity. Therefore, the previous structure needs to be flattened and extended in order to respond the requirement TS2 (Fig. 18).

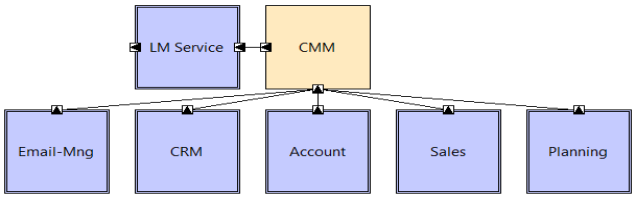

Figure 18. ADM_2: evolved architecture by applying flatten and extension

The CMM component was replaced by a new component (Enterprise Service Bus) responsible for managing service composition and message routing tasks and resource facilities. Moreover, the relevant functions from applications were exposed as software services. Therefore, the previous description of the architecture needs to be rewired, a component needs to be replaced, some ports need to be refined and the LM Service function needs to be abstracted away in new component ESB (Fig. 19).

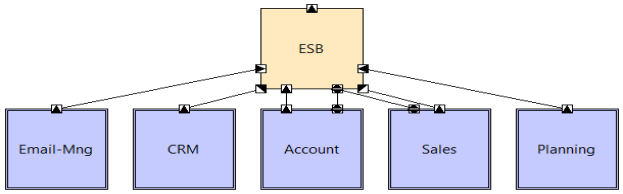

Figure 19. ADM_3: The evolved architecture by applying replacement, rewire, abstraction, and refinement

VI. Related Work

Change management is a critical problem in software engineering and specifically in software evolution research [18] that has been studied in a range of research domains.

Change management in business processes is a relatively mature area [9]. The ultimate goal of research on the evolution of workflow processes aims to enable business processes to evolve in a controlled and predictable manner. However, since this research track of change management focuses on business processes without taking software architecture that support its implementation into consideration, they are inadequate to support the multi-aspect nature of software architecture evolution. Although in [20] a set of change impact patterns are specified for capturing the types of change effect in services-based business processes and they were actually an inspiration for our change pattern definitions, they only focus on behavioral change impact without taking structural aspect of software architecture into account. Moreover, they have not laid a formal basis to perform analytical activities with the patterns instead they hardcoded them in a tool that accomplish some automated change impact analysis and propagation in service-based business processes.

Model-driven software architecture evolution community uses models as artifacts to describe well-defined software aspects at a higher abstraction level than source code. Model transformation is a well-established technique to modify and evolve models [2]. A well-known formalism used for model transformation is graph transformation, which enables reasoning about the formal properties of model transformations. Using model transformation, and especially graph transformation, to express and formalize the evolution of architectural descriptions is not a new approach towards software architecture evolution. Le Métayer [3] proposed such an approach in 1998. More recently, Grunske [4] formalized architectural reconfiguration as graph transformations that can be applied automatically. Tamzalit and Mens [5] used graph transformations to express architectural evolution patterns to introduce architectural styles as well as to verify whether a given architectural evolution preserves the constraints imposed by an architectural style. Another approach to transformation-based architectural evolution, though not directly relying on graph transformation, is work by Barais [6]. These contributions focus on structural aspects of architecture descriptions without taking the behavior into account. Moreover, they enable evolution of the architecture without considering other related artifacts as a driver of primary changes. While often the double push-out (DPO) construction is used for graph transformations, we follow [17] here and plan to further formalize our approach using mappings between locally surjective homomorphisms (LSH).

Change-driven transformation put forward the concept of change as a first class entity in model transformation. [21] is the closest to our contribution as both focus on change pattern for propagating changes in the model-to-model context. However, they focus on language aspect for specifying change-driven transformation while we look at concrete domain-specific instances of change patterns.

Model traceability. MDSE has adopted traceability support for a vast variety of applications in model-driven context especially when there is a need for analytical and automated generative support. Most of the contributions in this context have been cited in [13, 17, 22] and from which those that concentrated on evolution, focus on batch-based change propagation rather than change-driven approach. Even those approaches that proposed incremental live transformation such as [13] are too general to be considered

as a practical approach to evolve the technical and multi-aspect architecture-based models.

Model synchronization is closely related to model transformation and management. The existing general model synchronization frameworks are not a promising solution for this problem [19]. First, they require users to explicitly write synchronization logic to deal with each type of primary change and on each of the associated models. Second, the mapping function from business process to software architecture is inherently interleaved with the decisions regarding the potential information loss or gain related to different levels of the model's expressiveness.

VII. CONCLUSION

The above-mentioned approaches to change management in different domains concentrate only on either business processes or software architecture separately. Normally, business processes and software architectures are coupled with each other with complex dependencies between business processes and architectures. Change analysis and change reactions are difficult due to the possible complex dependencies between these two models. These dependencies have not been fully addressed. Our research presents an approach for filling the gaps.

Our work enables the co-evolution of business process and software architecture models. A comprehensive set of structural and behavioral change impact patterns are defined. These change patterns are the incremental evolution of software architecture models that can be reused in software evolution process. The identified change impact patterns are useful to track the history and reduce the complexity of changes. A graph-based formalism is provided and its implications are also discussed.

In the future, we fully formalize the change patterns. This will enable us to develop the management and analysis features further. We will complement these activities by an empirical investigation on selected case studies (beyond the ones used for initial change pattern determination and illustration with an especial focus on settings in which more than one business processes are related to an architecture model) to evaluate the comprehensiveness and usefulness of the proposed change pattern.

ACKNOWLEDGMENT

This research work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

REFERENCES

[1] T. Mens, J. Magee, B. Rumpe, "Evolving Software Architecture Descriptions of Critical Systems", IEEE Computer 43: 5, 42-48 May, 2010.

[2] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software, vol. 20, no. 5, pp. 42-45. 2004.

[3] D. Le Métayer. Describing Software Architecture Styles Using Graph Grammars. IEEE Trans. Software Eng., vol. 24, no. 7, pp. 521-533. 1998.

[4] L. Grunske. Formalizing Architectural Refactorings as Graph Transformation Systems. Proc. 6th Int'l Conf. Software Eng., Artificial Intelligence, Networking, and Parallel/Distributed Computing and 1st ACIS Int'l Workshop Self-Assembling Wireless Networks (SNPD/SAWN 05), IEEE CS Press, pp. 324-329. 2005.

[5] D. Tamzalit and T. Mens. Guiding Architectural Restructuring through Architectural Styles. Proc. 17th Ann. IEEE Int'l Conf. and Workshop Eng. of Computer-Based Systems (ECBS 10), IEEE Press, pp. 69-78. 2010.

[6] O. Barais, A. Le Meur, L. Duchien, and J. Lawall, "Software Architecture Evolution," Software Evolution, T. Mens and S. Demeyer, eds., Springer, pp. 233-262. 2008.

[7] K. Yskout, R. Scandariato, W. Joosen, Change patterns: Co-evolving requirements and architecture, Report CW593, August 2010.

[8] J. Gray, Y. Lin, and J. Zhang, Automating Change Evolution in Model-Driven Engineering, Computer 39, no. 2, 51–58. 2006.

[9] B. Weber, M. Reichert and S. Rinderle Ma. Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. Data and Knowledge Engineering, 66 (3), pp. 438-466. 2008.

[10] R. Heckel. Graph Transformation in a Nutshell. In: School on Foundations of Visual Modelling Techniques (FoVMT 2004). ENTCS, vol. 148(1), pp. 187–198. Elsevier, Amsterdam. 2006

[11] C. Costa Soria, R. Heckel: Modelling the Asynchronous Dynamic Evolution of Architectural Types. Proceedings Intl. Conference on Self-organizing Architectures SOAR 2009: 198-229. 2009.

[12] C. Tibermacine, R. Fleurquin, S. Sadou: A family of languages for architecture constraint specification. Journal of Systems and Software 83(5): 815-831. 2010.

[13] D. Hearnden. Deltaware: Incremental Change Propagation for Automating Software Evolution in the Model-Driven Architecture. Ph.D. Thesis. University of Queensland, 2007.

[14] S. Khoshnevis, P. Jamshidi, R. Teimourzadegan, A. Nikravesh, A. Khoshkbarforoushha, F. Shams. ASMEM: A Method for Automating Model Evolution of Service-Oriented Systems, Maintenance and Evolution of Service-Oriented Systems (MESOA) 2009.

[15] A. Ahmad and C. Pahl, Pat-Evol: Pattern-driven Reuse in Architecture-based Evolution for Service Software, ERCIM 88, 2012.

[16] H. Erdogmus. Representing architectural evolution. In: Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research, pp. 159–177. 1998.

[17] V. Gacitua-Decar, Graph-based Pattern Matching and Discovery for Process-centric Service Architecture Design and Integration, PhD Thesis, 2010.

[18] J. Estublier, D. Leblang, A. Hoek., R. Conradi, G. Clemm, W. Tichy and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. ACM Trans. Softw. Eng. Methodol. 14, 4 2005, 383-430. 2005.

[19] S. Buchwald, T. Bauer and M. Reichert. Bridging the gap between business process models and service composition specifications. Int'l Handbook on Service Life Cycle Tools and Technologies: Methods, Trends and Advances, 2011.

[20] Y. Wang, J. Yang, and W. Zhao. Change impact analysis for service based business processes. IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2010, pp. 1–8. 2010.

[21] G. Bergmann, I. R´ath, G. Varr´o, and D. Varr´o. "Change-driven model transformations. change (in) the rule to rule the change." Software and Systems Modeling, pp. 1–31, 2011.

[22] V. Gruhn, M. Wever, and C. Pahl. Data Model Evolution as Basis of Business. Process Management. International Conference on Object-Oriented and Entity Relationship Modelling O-O ER'95. 1995.